# **Data Retrieval**

## **Objectives**

**3.1 Construct and analyze queries that select data**

INNER JOIN, LEFT JOIN, RIGHT JOIN, CROSS JOIN (Cartesian product), and FULL OUTER JOIN; self joins; combine result sets by using UNION and INTERSECT; DISTINCT; column alias; computed columns

**3.2 Construct and analyze queries that sort and filter data**

ORDER BY, WHERE, LIKE, BETWEEN, AND, OR, NOT, TOP (LIMIT), IN, NOT IN, ANY, ALL, NULL, NOT NULL, comparison operators

**3.3 Construct and analyze queries that aggregate data**

GROUP BY, HAVING, MIN, MAX, COUNT, AVG (AVERAGE), SUM

## **Reading Materials**

SQL Primer - An Accelerated Introduction to SQL Basics: Chapters 5, 8, 9, 10, 11, 12 & 13

### Chapter 5: Writing Basic Queries

#### Overview

A *query* is an SQL statement that is used to extract a subset of data from your database and presents it in a readable format. As we have seen previously, the SELECT command is used to run queries in SQL. You can further add clauses to your query to get a filtered, more meaningful result. The level of flexibility afforded by SQL is one of the reasons it has succeeded as a query language. While there is an entire gamut of add-ons to SELECT, in this chapter we will focus on only two – **ORDER BY and WHERE.**

Database administration tasks for a well thought-out schema are few and far between, but retrieving meaningful results using queries is something everyone does routinely. Since the majority of operations on a database involve queries, it is important to understand them in detail. While this chapter will only deal with queries run on a single table, you can run a SELECT operation on multiple tables in a single statement.

Selecting a Limited Number of Columns

The intention since the beginning of SQL was to provide an easy-to-use query system to everyday users. They should not have to reach for a programming language to make their report readable. A major facility for this is the ability to display a finite set of columns in the output rather than all the fields of a table.

We have already seen how to extract all the data from a table when we were verifying our results in the previous chapters. But as you might have noted in some of our catalog queries – we can extract a subset of data too. We first test this by limiting the number of fields to show in the query output by not specifying the \* selection criteria, but by naming the fields explicitly as a comma-separated list ([Listing 5-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=671695994&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl;

| language | year |
| --- | --- |
| Prolog | 1972 |
| Perl | 1987 |
| APL | 1964 |

You can see that the query we constructed mentioned the fields we wish to see, that is, *language* and *year*. Also note that the result of this query is useful by itself as a report for looking at the chronology of programming language creation. While this is not a rule enforced by SQL or a relational database management system, it makes sense to construct your query in such a way that the meaning is self-evident if the output is meant to be read by a human. This is the reason we left out the field *id* in the query, since it has no inherent meaning to the reader except if they wish to know the sequential order of the storage of records in the table.

You are free to decide the ordering of the fields in your output. The positioning of a field in a CREATE TABLE statement has no effect on any SELECT query you run on it. Indeed, you are even free to duplicate a field as many times as you wish in your output. Whether it makes sense to do so is debatable! But as long as the field names in the comma-separated list to SELECT is valid, it will show up in the output.

Ordering the Results

You might have noticed that in our previous query output, the languages were printed out in the same order as we had inserted them. But what if we wanted to sort the results by the year the language was created in. The chronological order might make more sense if we wish to view the development of programming languages through the decades. In such cases, we take the help of the **ORDER BY** clause. To achieve our purpose, we modify our query with this additional clause ([Listing 5-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=816026320&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl ORDER BY year;

| language | year |
| --- | --- |
| APL | 1964 |
| Prolog | 1972 |
| Perl | 1987 |

The astute reader will notice that the output of our ORDER BY clause was ascending. This is the default ordering that can be made explicit by appending the argument ASC to the column we wish to sort. To reverse this, we use the argument DESC to our ORDER BY clause as below ([Listing 5-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=816026320&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl ORDER BY year DESC;

| language | year |
| --- | --- |
| Perl | 1987 |
| Prolog | 1972 |
| APL | 1964 |

Ordering is not limited to numeric fields. You can order character-based columns too. The sorting method is alphabetical starting with the first character and subsequently moving to the next sequential characters if the character is the same. Let us try ordering our query result by the *language* field this time ([Listing 5-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=816026320&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl ORDER BY language;

| language | year |
| --- | --- |
| APL | 1964 |
| Perl | 1987 |
| Prolog | 1972 |

Ordering using Field Abbreviations

A useful shortcut in SQL involves ordering a query result using an integer abbreviation instead of the complete field name. The abbreviations are formed starting with 1, which is given to the first field specified in the query; 2 to the second field; and so on. Let's rewrite our query to sort the output by descending year using field abbreviations ([Listing 5-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=250665784&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl ORDER BY 2 DESC;

| language | year |
| --- | --- |
| Perl | 1987 |
| Prolog | 1972 |
| APL | 1964 |

The 2 argument given to the ORDER BY clause signifies ordering by the second field specified in the query, namely *year*. Over time I have realized that the best use of field abbreviations is while you are querying a database system interactively. Rarely is it a good idea to use field abbreviations if you are embedding SQL inside a programming language.

Ordering by Multiple Columns

What if you wanted to order your results by more than one column? It would be a plausible scenario where some of the values of the ordering column are the same. For example, supposing you had a table having student grades and names. You want to order the students by their grades, but a lot of students have gotten the Grade A. So you apply a second ordering by name, sorting alphabetically all grade A students, then grade B students and so on.

Let's try to see a working example of this using our programming languages table. But for that we need to insert a few more rows in there ([Listing 5-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=896491260&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tbl (id, language, author, year, standard) VALUES

(4, 'JOVIAL', 'Schwartz', 1959, 'US-DOD');

INSERT INTO proglang\_tbl (id, language, author, year, standard) VALUES

(5, 'APT', 'Ross', 1959, 'ISO');

Now let us order our programming languages table by *year* and *language* keeping in mind that our newly inserted languages have the same year of creation ([Listing 5-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=896491260&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl ORDER BY year, language;

| language | year |
| --- | --- |
| APT | 1959 |
| JOVIAL | 1959 |
| APL | 1964 |
| Prolog | 1972 |
| Perl | 1987 |

You can even use different ordering types for each of the columns ([Listing 5-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=896491260&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM proglang\_tbl ORDER BY year DESC, language ASC;

| language | year |
| --- | --- |
| Perl | 1987 |
| Prolog | 1972 |
| APL | 1964 |
| APT | 1959 |
| JOVIAL | 1959 |

Notice how *APT* came before *JOVIAL* because we had mentioned an ascending order for the *language* field.

Putting Conditions with WHERE

We have already seen how to select a subset of data available in a table by limiting the fields queried. We will now limit the number of records retrieved in a query using conditions. The WHERE clause is used to achieve this, and it can be combined with explicit field selection or ordering clauses to provide meaningful output.

For a query to run successfully and fetch data from a table, it must have at least two parts – the SELECT and the FROM clause After this we place the optional WHERE condition and then the ordering clause. Thus, if we wanted to see the programming language (and its author), which was standardized by ANSI, we'd write our query as below ([Listing 5-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=491886073&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, author FROM proglang\_tbl WHERE standard = 'ANSI';

| language | author |
| --- | --- |
| APL | Iverson |

As you may have noticed, the query we formulated specified the *language* and *author* fields, but the condition was imposed on a separate field altogether – *standard*. Thus we can safely say that while we can choose what columns to display, our conditionals can work on a record with any of its fields.

You are by no means restricted to use = (equals) for your conditions. It is perfectly acceptable to choose other operators like < and >. You can also include the ORDER BY clause and sort your output. An example is given below ([Listing 5-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=491886073&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, author, year FROM proglang\_tbl WHERE year > 1970 ORDER BY author;

| language | author | year |
| --- | --- | --- |
| Prolog | Colmerauer | 1972 |
| Perl | Wall | 1987 |

Notice that the output only shows programming languages developed after 1970 (at least according to our database). Also since the ordering is done by a varchar field, the sorting is done alphabetically in an ascending order.

If we let go of our from a table requirement, we can write a query with just SELECT. Try SELECT 1 in your DBMS and see the output.

Combining Conditions

If we can only specify one condition using the WHERE clause, it will fulfill only a tiny fraction of real-world requirements. We can however construct complex conditions using the *boolean* operators AND & OR.

When we want our resultset to satisfy all of the multiple conditions, we use the AND operator ([Listing 5-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=131554793&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, author, year FROM proglang\_tbl

WHERE year > 1970 AND standard IS NULL;

| language | author | year |
| --- | --- | --- |
| Perl | Wall | 1987 |

We have now combined the two conditions, meaning any row in the resultset must satisfy both the criteria mentioned. In our case, there is only one such row – *Perl*.

An interesting point to note is our construction of the second conditional. We specify that the *standard* field should be a null value by specifying IS NULL. This is not the same as saying standard = NULL. If we attempt to write the latter as our conditional, we would get an empty result.

While this may seem counter-intuitive, it actually makes perfect sense. A null is supposed to signify undefined values, not a precise value like infinity or 0 or even a complex number. We cannot rationalize the precise equivalence operator = for a null, and thus SQL interpreters use the IS NULL comparison.

If we want our resultset to satisfy any one of our conditions, we use the OR operator. Let's use this operator in the above example but with a different intention. We want all languages that were either created after 1970 *or* don't have a standardizing body ([Listing 5-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=131554793&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, author, year FROM proglang\_tbl

WHERE year > 1970 OR standard IS NULL;

| language | author | year |
| --- | --- | --- |
| Prolog | Colmerauer | 1972 |
| Perl | Wall | 1987 |

*Prolog* only satisfies the first criterion that it was created after 1970 but was actually standardized by ISO. *Perl* satisfies both criteria and is also rightly shown. If we had a language in our table without a standardizing body but created before 1970, it would also sneak up on the resultset here.

We can even create yet more complex queries by combining the AND & OR operators. One has to be careful to not make the logic of the filtering using these operators complex or unreadable.

### Chapter 8: Doing More with Queries

#### Overview

SQL as a language was created for the end users of the database systems. It just happened to be used by programmers too, but the goal was always a simple, declarative, English-like language to allow anybody familiar with computers and the domain to make sensible reports out of the database system. These reporting capabilities were the direct output of an SQL query, and thus from the very beginning, there have been a lot of options and clauses that can be used with SELECT to make the output more legible.

We have already seen some basic queries, how to order the results of a query, and how to put conditions on the query output. Let us now see more examples of how we can modify our SELECT statements to suit our ever-growing reporting needs.

Counting the Records in a Table

Sometimes we just wish to know how many records exist in a table without actually outputting the entire contents of these records. This can be achieved through the use of an SQL *junction* called COUNT. Let us first see the contents of the *proglang\_tbl* table when we last left it ([Table 8-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=528677597&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 3 | APL | Iverson | 1964 | ANSI |
| 4 | Jovial | Schwartz | 1959 | US-DOD |
| 5 | APT | Ross | 1959 | ISO |
| 7 | TcI | Ousterhout | 1988 |  |

We can clearly see that using the *id* of the last row of the table, 7 in this case, is clearly not a good idea. While there may have been 7 rows in the table at some point in the past, we had actually deleted the Forth language row. Additionally we cannot always rely on such a field, especially when we could have inserted an *id* of 4711 in the field without anybody complaining. Clearly, we need COUNT to come to our rescue ([Listing 8-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=528677597&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT COUNT(\*) FROM proglang\_tbl;

The output returned will be a single record with a single field with the value as 6. The function COUNT took one argument, that is, what to count and we provided it with \* that means the entire record. Thus we achieved our purpose of counting records in a table.

What would happen if instead of giving an entire record to count, we explicitly specify a column? And what if the column had null values? Let's see this scenario by counting on the *standard* field of the table ([Listing 8-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=528677597&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT COUNT(standard) FROM proglang\_tbl;

If you guessed the output of this query as the value 4, you are correct. Out of the six rows, two records contain null as their *standard* value, leaving out four languages with a *standard*.

Using DISTINCT with COUNT

The astute reader might have noticed that the number of standardized languages was computed by counting the number of non-null *standard* values. However the resultset contained a duplicate standards body value – *ISO* for both APT and Prolog.

Sometimes it is useful to be able to leave out such duplicates. The DISTINCT clause allows us to utilize only non-duplicated values of the input specified and is commonly used in conjunction with COUNT. Before seeing it in action, let's add another row to our table so that the results of using DISTINCT jump out clearly ([Listing 8-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=805905979&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tbl (id, language, author, year, standard) VALUES

(6, 'PL/I', 'IBM', 1964, 'ECMA');

Note the new data choice that we are populating with our new row. With PL/I, we now have a fourth distinctive standards organization – ECMA. PL/I also shares the same birth year as APL (1964) giving us a duplicate *year* field. Now let us run a query to check what distinct *year* values we have in the table ([Listing 8-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=805905979&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT DISTINCT year FROM proglang\_tbl;

| Year |
| --- |
| 1972 |
| 1988 |
| 1987 |
| 1964 |
| 1959 |

Both 1964 and 1959 make an appearance only once as we desired. A common use case for DISTINCT is to combine it with the COUNT function to output the number of unique values we have in the table ([Listing 8-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=805905979&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Attempting the same for *year*, we get our expected result of 5.

SELECT COUNT (DISTINCT year) FROM proglang\_tbl;

> 5

Using DISTINCT on the standard field has a slightly different output than we might expect at first guess ([Listing 8-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=805905979&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT DISTINCT standard FROM proglang\_tbl;

| standard |
| --- |
| (null) |
| ECMA |
| ANSI |
| ISO |
| US-DOD |

We actually get five rows in our output including the null value because for the DISTINCT clause, it is a uniquely separate value. Combining this with COUNT removes the significance of the null row giving us the value 4 ([Listing 8-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=805905979&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT COUNT (DISTINCT standard) FROM proglang\_tbl;

> 4

Column Aliases

Queries are frequently consumed directly as reports, since SQL provides enough functionality to give a meaningful representation to the data stored inside a RDBMS. One of the features allowing this is *Column Aliases*, which let you rename column headings in the resultant output. The general syntax for creating a column alias is given below ([Listing 8-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=367317825&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT <column1> <alias1>,

<column2> <alias2>,

...

FROM <table>;

For example, we wish to output our programming languages table with a few columns only. But we do not wish to call the authors of the language as *authors*. The person wanting the report wishes they be called *creators*. This can be simply done by using the query below ([Listing 8-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=367317825&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT id, language, author creator FROM proglang\_tbl;

| id | language | creator |
| --- | --- | --- |
| 1 | Prolog | Colmerauer |
| 2 | Perl | Wall |
| 3 | APL | Iverson |
| 4 | JOVIAL | Schwartz |
| 5 | APT | Ross |
| 7 | TcI | Ousterhout |
| 6 | PL/I | IBM |

While creating a column alias will not permanently rename a field, it will show up in the resultant output. Implementations differ on whether they allow column aliases to be used in other parts of the query other than column listing. For example, let's try using the column alias *creator* in the WHERE clause of a query in PostgreSQL ([Listing 8-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=367317825&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT id, language, author creator

FROM proglang\_tbl WHERE creator = 'Ross';

ERROR: column "creator" does not exist

LINE 4: FROM proglang\_tbl WHERE creator = 'Ross';

Aha, PostgreSQL explicitly told us that this is a no-go. Let's see if SQLite is slightly more forgiving ([Listing 8-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=367317825&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> SELECT id, language, author creator

FROM proglang\_tbl WHERE creator = 'Ross';

id language creator

---------- ---------- ----------

5 APT Ross

While SQLite did allow it, I'm not really fond of using column aliases in anything other than column renaming for the output. I'd advise that you do the same unless there is a very strong case of readability improvement and your implementation allows it (and there are very few that allow it).

Order of Execution of SELECT Queries

A query is not evaluated from left to right; there is a specific sequence in which its various parts are evaluated as given below.

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

There is an interesting corollary of having the SELECT evaluation being lower than the WHERE clause. Can you guess what it is?

It is the inability of database management systems like PostgreSQL, DB2, and Microsoft SQL Server to use column aliases in the WHERE clause. Until the point the query execution is on the filtering stage using the conditions provided, it has still not resolved the column aliases of the query.

Let's test this by running a query in PostgreSQL where we use the column alias in the ORDER BY clause, the only one with a lower precedence ([Listing 8-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=398586733&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# SELECT id, language, author creator

FROM proglang\_tbl ORDER BY creator;

id | language | creator

----+----------+------------

1 | Prolog | Colmerauer

6 | PL/I | IBM

3 | APL | Iverson

7 | Tcl | Ousterhout

5 | APT | Ross

4 | JOVIAL | Schwartz

2 | Perl | Wall

(7 rows)

Our reasoning was rewarded with the correct output, not that I would change my advice of using column aliases only with proper thought.

Using the LIKE Operator

While putting conditions on a query using WHERE clauses, we have already seen comparison operators = and IS NULL. Now we take a look at the LIKE operator, which will help us with wildcard comparisons. For matching we are provided with two wildcard characters to use with LIKE.

**% (Percent):** used to match multiple characters, including a single character and no character

**\_ (Underscore):** used to match exactly one character

We will first use the % character for wildcard matching. Let us suppose we wish to list out languages that start with the letter **P** ([Listing 8-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=610833220&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tbl WHERE language LIKE 'P%';

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 6 | PL/I | IBM | 1964 | ECMA |

The output of the above query is all language records whose name begins with the letter capital P. While we don't have such a record, note that this resultset would not include any language that starts with the small letter p.

We can see that using the % wildcard allowed us to match multiple characters like *erl* in the case of Perl. But what if we wanted to restrict how many characters we wished to match? What if our goal was to write a query that displays the languages ending in the letter l, but are only three characters in length? The first condition could have been satisfied using the pattern %l, but to satisfy both conditions in the same query we use the \_ wildcard. A pattern like %l would result in returning both *Perl* and *Tcl*, but we modify our pattern suitably to return only the latter ([Listing 8-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=610833220&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tbl WHERE language LIKE '\_\_l';

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 7 | Tcl | Ousterhout | 1988 |  |

Note that the result did not include *Perl* since we explicitly gave two underscores to match two characters only. Also it did not match *APL* or *JOVIAL* since SQL data is case sensitive and l is not equal to L.

We can also use NOT in conjunction with LIKE to negate or inverse the result. If we used a NOT in the conditional clause of [Listing 8-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=610833220&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#), what languages do we expect to get back in the result? Having *Perl, APL*, and *JOVIAL* is certainly right, but they are not the entire resultset. Any language that is not three characters long and ending with a lowercase l would be in the output ([Listing 8-15](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=610833220&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tbl WHERE language NOT LIKE '\_\_l';

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 3 | APL | Iverson | 1964 | ANSI |
| 4 | JOVIAL | Schwartz | 1959 | US-DOD |
| 5 | APT | Ross | 1959 | ISO |
| 6 | PL/I | IBM | 1964 | ECMA |

Be careful when using LIKE; its comparisons are computationally expensive on the database, especially the ones involving multiple % wildcards.

### Chapter 9: Calculated Fields

#### Overview

We have already seen *column aliases* that allow us to rename a field's name in the query output. But we frequently encounter conditions that require changes to a field value. This is where the concept of a *calculated field* comes in.

Mathematical Calculations

Any numeric field can be operated upon by mathematical operators we are all familiar with. We can add, subtract, multiply, divide, and even find the remainder of a division operation fairly easily. While the operators supported differ in various implementations, the ones given below should be available across any RDBMS you come across ([Table 9-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=565067362&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

| Addition | + |
| --- | --- |
| Subtraction | - |
| Multiplication | \* |
| Division | / |
| Remainder | % |

Let us take our programming languages table and try to find out the decade in which the language was created. For example, Prolog was created in the 1970s decade. Let us try to find out this fact from the year of creation available to us. One approach is to find the remainder of the year when divided by 10, which is the number of years in a decade ([Listing 9-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=565067362&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This is the value that specifies how many years it has been since the start of that decade.

SELECT language, (year % 10) remain FROM proglang\_tbl;

| language | remain |
| --- | --- |
| Prolog | 2 |
| Perl | 7 |
| APL | 4 |
| JOVIAL | 9 |
| APT | 9 |
| Tcl | 8 |
| PL/I | 4 |

Now if we subtract this value from the year of creation itself, we would get the decade in which the programming language was created ([Listing 9-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=565067362&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year – (year % 10) decade FROM proglang\_tbl;

| language | decade |
| --- | --- |
| Prolog | 1970 |
| Perl | 1980 |
| APL | 1960 |
| JOVIAL | 1950 |
| APT | 1950 |
| Tcl | 1980 |
| PL/I | 1960 |

Another approach is to divide the year by 10 and then multiply it by 10. This is slightly less straightforward because it relies on the definition of the integer data type. Since an integer cannot store decimal points, division by 10 would silently chop off the remainder. The year 1972 divided by 10 would be 197 discarding the .2 bit. If we multiply this value by 10, we would get our desired decade value ([Listing 9-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=565067362&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, (year / 10) \* 10 decade FROM proglang\_tbl;

String Operations

By far the most commonly used string operation is *concatenation*. It means to join or combine strings. However, since even numeric fields can be treated as a string, we can use the concatenation operator || on them too. See the example below to modify our *decade* field to include some characters ([Listing 9-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=406456441&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, 'The '||((year/10)\*10)||'s' decade FROM proglang\_tbl;

| language | decade |
| --- | --- |
| Prolog | The 1970s |
| Perl | The 1980s |
| APL | The 1960s |
| JOVIAL | The 1950s |
| APT | The 1950s |
| Tcl | The 1980s |
| PL/I | The 1960s |

Note that the concatenation operator manifests itself in different forms in different implementations. PostgreSQL, SQLite, and Oracle use the shown || symbols whereas Ingres, MySQL, and Microsoft SQL Server use + to denote concatenation. Their effect, however, is the same.

The string concatenation operator differs in various programming languages too. The **||** character, which is used in most SQL implementations, comes from the IBM PL/I – a language quite popular in the '60s and '70s but rarely seen in modern times.

The last character I actually is the roman numeral for 1, as it was built purposely to unify the growing gap between languages specializing in business processes and those catering to scientific computation.

Another common string operation is *substring*, which returns only a part of the string field value. For example, if we needed to get only the first two characters of each programming language, we would use the SUBSTR function. The general syntax of this function is given below ([Listing 9-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=406456441&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT SUBSTR(<field name>, <starting position>, <length>),

…

FROM <table>;

The *starting position* is the character you wish to start extracting from. Unlike most programming languages, the string index positions here don't start from 0 but 1. The third argument *length* specifies how many characters should be a part of the result. For the first two characters of a programming language, the starting position would be 1 and the length would be 2 ([Listing 9-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=406456441&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT SUBSTR(language, 1, 2), year FROM proglang\_tbl;

| substr | year |
| --- | --- |
| Pr | 1972 |
| Pe | 1987 |
| AP | 1964 |
| JO | 1959 |
| AP | 1959 |
| Tc | 1988 |
| PL | 1964 |

Interestingly, on one hand, PostgreSQL gives the same result when we use SUBSTRING instead of SUBSTR, because it treats them as aliases. SQLite, however, works only with SUBSTR. Microsoft SQL Server, on the other hand, works only with SUBSTRING. Check your database manual for which version your implementation expects.

Another class of string operations that often comes in handy are UPPER and LOWER, which change the case of a string value to upper- and lowercase respectively. This is best illustrated with an example ([Listing 9-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=406456441&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#))

SELECT UPPER(language), LOWER(standard) FROM proglang\_tbl;

| upper | lower |
| --- | --- |
| PROLOG | iso |
| PERL |  |
| APL | ansi |
| JOVIAL | us-dod |
| APT | iso |
| TCL |  |
| PL/I | ecma |

Literal Values

There are cases when one needs to use a fixed literal value as the values of a new column. Like column aliases can change the column header for readability, literal values change record values. In a sense they are not calculated fields, but fixed fields inserted in specific positions of a record. An example will help illustrate this – supposing you wish to really clarify the year of language creation, as not just a number but also to include the characters AD ([Listing 9-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=306413202&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year, 'AD' FROM proglang\_tbl;

| language | year | AD |
| --- | --- | --- |
| Prolog | 1972 | AD |
| Perl | 1987 | AD |
| APL | 1964 | AD |
| JOVIAL | 1959 | AD |
| APT | 1959 | AD |
| Tcl | 1988 | AD |
| PL/I | 1964 | AD |

We can even use numeric literal values the same way, omitting the quotation marks for such values. A common utility for literal values arises when the user has to copy-paste data from their database query output into another tool like a spreadsheet or word processor.

### Chapter 10: Aggregation and Grouping

#### Overview

SQL has maintained its prominent position in the technical world due to its ability to cater to a wide range of business intelligence and analytic requests. While databases are often used for finding a needle in a haystack, that is, narrowing down to a single row, a lot of interactive usage of SQL revolves around generating aggregated insights from a bunch of rows.

Indeed, a major advantage that SQL-based systems have over NoSQL data storage solutions is how intuitive grouping and aggregation is in the former category.

Aggregate Functions

An *aggregate function* is used to compute summarization information from a table or tables. We have already seen the COUNT aggregate function that counts the records matched. Similarly, there are other aggregation functions in SQL like AVG for calculating averages; SUM for computing totals; and the extreme functions MAX, MIN for finding out maxima and minima values respectively.

The count and extreme functions work with all data types, but functions like SUM and AVG make sense only with numeric types and thus work only with them.

Let's now try to use AVG on the only sensible numeric choice we have in our table – *year* ([Listing 10-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=296850498&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). You can think of the below query as a way of finding out the mean year value of all the programming language records we have in our table.

1. SELECT AVG(year) FROM proglang\_tbl;

| avg |
| --- |
| 1970.4285714285714286 |

We can see the result as a decimal number with a default of 16 digits after the decimal point in PostgreSQL. This is slightly lowered to 10 digits after the decimal point in SQLite but more than enough still to cover all but the rare scenarios.

While the average value was calculated accurately, one can argue that a value like this to specify *year* is not useful. What we really want is a readable year value that looks like an actual year, an integer value specifically. Thus we go about *casting* this average value to an integer ([Listing 10-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=296850498&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT CAST(AVG(year) AS INTEGER) FROM proglang\_tbl;

| avg |
| --- |
| 1970 |

Conversion of data types using CAST only works with compatible data types like numerics and integers. If you try to convert a *varchar* into an *integer*, the DBMS will spit out an error message as is proper ([Listing 10-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=296850498&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# SELECT CAST(language AS INTEGER) FROM proglang\_tbl;

ERROR: invalid input syntax for integer: "Prolog"

The exception here is SQLite, which will obediently convert the *varchar* value to 0. This may be the most sensible choice for it because of its underlying engine implementation; however, I would warn you to stay away from such surprising cast operations.

Let's try the other numeric aggregate function that is commonly used – SUM. Suppose we wish to find the sum of the *year* values in our table – the query would be written in a straightforward way using SUM ([Listing 10-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=296850498&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT SUM(year) FROM proglang\_tbl;

| sum |
| --- |
| 13793 |

As before, if we use SUM on a varchar field, PostgreSQL would spit out an error while SQLite would quietly give a value of 0.0 ([Listing 10-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=296850498&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# SELECT SUM(language) FROM proglang\_tbl;

ERROR: function sum(character varying) does not exist

LINE 1: SELECT SUM(language) FROM proglang\_tbl; ^

HINT: No function matches the given name and argument types.

You might need to add explicit type casts.

Using the Extreme Functions - MAX and MIN

MAX and MIN are collectively called the extreme functions because they essentially find the extreme values from a set of column values. Their most intuitive application is with numeric data, but these functions can be applied to other database types as well.

These functions are fairly straightforward to understand and use. Let's take the MIN first. It looks at a particular set of rows and finds the minimum value of the column that is provided as an argument to it. For example, in our table we wish to find out from which year do we have records of programming languages, that is, the earliest language year. Analyzing the problem at hand, we see that if we apply the aggregate function MIN to the field *year* in our table, we should get the desired output ([Listing 10-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=673425581&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT MIN(year) FROM proglang\_tbl;

| min |
| --- |
| 1959 |

We had two languages in our table corresponding to the year 1959 – *APT* and *JOVIAL*. But since this is the minimum value, it was the result once regardless of how many languages had the same value. The function MAX is similar, but its result would signify the latest year in which one of the languages in our table was created ([Listing 10-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=673425581&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT MAX(year) FROM proglang\_tbl;

| max |
| --- |
| 1988 |

Of course you can combine them in a single query to get the result in a single row itself ([Listing 10-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=673425581&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT MAX(year), MIN(year) FROM proglang\_tbl;

| max | min |
| --- | --- |
| 1988 | 1959 |

Like we read before, these functions are not limited to numeric types. So let's combine finding the MAX *year* value with the MIN *language* value ([Listing 10-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=673425581&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT MAX(year), MIN(language) FROM proglang\_tbl;

| max | min |
| --- | --- |
| 1988 | APL |

The MIN language found was *APL* since it's the first alphabetically. Notice that *APT* was not chosen since L < T when comparing alphabets.

We need to be careful while reading this result though. At first glance it gives a misleading view that *1988* corresponds to *APL*, which is not the case. Our query simply gives the extreme values of these two fields in our table, whether they are from the same record or not.

Grouping Data

The GROUP BY clause of a SELECT query is used to *group records* based upon their field values. This clause is placed after the WHERE conditional. For example, in our sample table we can group data by which committee decided on their standard. Let's see this action after we insert another record in our table so that the logical nature of grouping becomes clearer ([Listing 10-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=180795054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tbl (id, language, author, year, standard) VALUES

(8, 'Fortran', 'Backus', 1957, 'ANSI');

SELECT standard FROM proglang\_tbl WHERE standard IS NOT NULL GROUP BY standard;

| standard |
| --- |
| ECMA4 |
| ANSI |
| ISO |
| US-DOD |

Notice how the different standards are *grouped* into a single value, regardless of how many times they occur in the table. Let's try to add the *language* column to the output of the above query ([Listing 10-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=180795054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard, language FROM proglang\_tbl

WHERE standard IS NOT NULL GROUP BY standard;

ERROR: column "proglang\_tbl.language" must appear in the GROUP BY clause or be used in an aggregate function

The database engine gave us an error for this query. This makes sense because while it bunched the different standards together because of our grouping clause, which language it should choose to display with it is ambiguous. Let us take the error message's first suggestion and also include the *language* field in the GROUP BY clause ([Listing 10-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=180795054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard, language FROM proglang\_tbl

WHERE standard IS NOT NULL GROUP BY standard, language;

| standard | language |
| --- | --- |
| ECMA | PL/I |
| ANSI | APL |
| US-DOD | JOVIAL |
| ISO | Prolog |
| ISO | APT |
| ANSI | Fortran |

The interesting thing to note here is the rule that the columns listed in the SELECT clause must be present in the GROUP BY clause. This leads us to the following two corollaries.

1. You cannot group by a column that is not present in the SELECT list.
2. You must specify all the columns in the grouping clause that are present in the SELECT list.

Bare Columns in SQLite

If you tried executing [Listing 10-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=180795054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) in SQLite, you would not get an error. This is because of the *bare column* feature in SQLite that allows a column to be present without aggregation in the SELECT clause and still be absent from the GROUP BY clause.

This is quite different from most DBMS systems out there, and i would suggest that you stay away from using this feature seriously because of the undefined behavior involved. However, it is good to read up on the details present at the SQLite website.

<https://www.sqlite.org/lang_select.html>

Grouping and Aggregate Functions

Another useful way to use grouping is to combine the operation with an aggregate function. Suppose we wish to count how many standards a particular organization has in our table. This can be achieved by combining the GROUP BY clause with the COUNT aggregate function as given below ([Listing 10-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=156500790&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard, COUNT(\*) FROM proglang\_tbl GROUP BY standard;

| standard | count |
| --- | --- |
| <null> | 2 |
| ECMA | 1 |
| ANSI | 2 |
| ISO | 2 |
| US-DOD | 1 |

The output is intuitive enough to warrant no further explanation, but the query itself is interesting. Notice that the GROUP BY clause consisted of only the *standard*. The aggregate function is a result of the bunching of the grouped columns.

Grouping truly makes sense in SQL when used judiciously with aggregate functions. A lot of utility and intelligence from databases are derived from analysts using a combination of these applied on a well-designed model. Let's see another example of combining GROUP BY with multiple aggregate functions this time.

Suppose we wish to find out how many languages were made in the same year, and of those languages which come first alphabetically ([Listing 10-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=156500790&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). We can immediately see that a GROUP BY on *year* is needed here along with a couple of different aggregate functions.

SELECT year, MIN(language), COUNT(\*) FROM proglang\_tbl GROUP BY year;

| year | min | count |
| --- | --- | --- |
| 1972 | Prolog | 1 |
| 1957 | Fortran | 1 |
| 1988 | Tcl | 1 |
| 1987 | Perl | 1 |
| 1964 | APL | 2 |
| 1959 | APT | 2 |

The HAVING Clause

Like a WHERE clause places conditions on the fields of a query, the HAVING clause places conditions on the groups created by GROUP BY. It must be placed immediately after the GROUP BY but before the ORDER BY clause ([Listing 10-15](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=447820761&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, standard, year FROM proglang\_tbl

GROUP BY standard, year, language HAVING year < 1980;

| language | standard | year |
| --- | --- | --- |
| APT | ISO | 1959 |
| JOVIAL | US-DOD | 1959 |
| APL | ANSI | 1964 |
| Fortran | ANSI | 1957 |
| PL/I | ECMA | 1964 |
| Prolog | ISO | 1972 |

From the output we can clearly see that the records for Perl and Tcl are left out since they do not satisfy the HAVING conditional of being created before 1980.

You might wonder why we need two different filtering clauses – WHERE and HAVING. A WHERE clause does not allow aggregate functions in its conditionals, a prime target for the HAVING clause. For example, suppose we wish to check which *standard* values exist more than once in our table. Our first stab at this using the GROUP BY clause might look something like this ([Listing 10-16](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=447820761&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard FROM proglang\_tbl WHERE COUNT(standard) > 1 GROUP BY standard;

ERROR: aggregate functions are not allowed in WHERE

Just like we thought, our SQL interpreter did not allow such a travesty. Instead we'll use the same conditional in the HAVING clause ([Listing 10-17](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=447820761&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard FROM proglang\_tbl GROUP BY standard HAVING COUNT(standard) > 1;

| standard |
| --- |
| ANSI |
| ISO |

It correctly gave us the names of the two *standard* values with more than one occurrence. Interestingly, if we tweak the conditional to COUNT(\*), we get an additional row ([Listing 10-18](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=447820761&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard FROM proglang\_tbl GROUP BY standard HAVING COUNT(\*) > 1;

| standard |
| --- |
| (null) |
| ANSI |
| ISO |

The filtering clause is now not restricted to non-null values of the *standard* column. Since there are multiple records with null values in the field, it will also be included in the result.

### Chapter 11: Understanding Joins

#### Overview

A *join* operation allows you to retrieve data from multiple tables in a single SELECT query. Two tables can be joined by a single join operator, but the result can be joined again with other tables. There must exist a same or similar column between the tables being joined.

When you design an entire database system using good design principles like *normalization*, we often require the use of joins to give a complete picture to a user's query. For example, we split our programming languages table into two in [Chapter 7](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=203#203) – one holding the author details ([Table 11-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=313284352&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)) and the other holding information about the languages itself ([Table 11-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=313284352&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). To show a report listing authors and which programming language they created, we would have to use a join.

| id | language | year | standard |
| --- | --- | --- | --- |
| 1 | Prolog | 1972 | ISO |
| 2 | Perl | 1987 | (null) |
| 3 | APL | 1964 | ANSI |
| 4 | Tcl | 1988 | (null) |
| 5 | BASIC | 1964 | ANSI |

| author\_id | author | language\_id |
| --- | --- | --- |
| 1 | Colmerauer | 1 |
| 2 | Wall | 2 |
| 3 | Ousterhout | 4 |
| 4 | Iverson | 3 |
| 5 | Kemeny | 5 |
| 6 | Kurtz | 5 |

We now form a query to show our desired output – the list of all authors with the corresponding language they developed ([Listing 11-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=313284352&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). We choose our join column as the *language\_id* field from the authors table. This corresponds to the *id* field in the languages table.

SELECT author, language FROM authors\_tbl, newlang\_tbl WHERE language\_id = id;

| author | language |
| --- | --- |
| Colmerauer | Prolog |
| Wall | Perl |
| Ousterhout | APL |
| Iverson | Tcl |
| Kemeny | BASIC |
| Kurtz | BASIC |

The output of our query combines a column from both tables giving us a better report. The language\_id = id is called the *join condition*. Since the operator used in the join condition is an equality operator (=), this join is called an *equijoin*. Another important thing to note is that the columns participating in the join condition are not the ones we choose to be in the result of the query.

Remember that the joining of tables to view a resultset does not affect the tables at all. Nothing physically changes in the tables themselves with respect to their structure or data. The implicit connection forming is only within the lifetime of the join query execution.

Alternative Join Syntax

You would have noticed that we formed our join query without much special syntax, using our regular FROM/WHERE combination. The SQL-92 standard introduced the JOIN keyword to allow us to form join queries. Since it was introduced earlier, the FROM/WHERE syntax is still quite popular for joins. But now that the majority of database vendors have implemented most of the SQL-92 standard, the JOIN syntax is also in widespread use. Below is the JOIN syntax equivalent of the query we just wrote to display which author created which programming language ([Listing 11-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=534647669&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT author, language FROM authors\_tbl JOIN newlang\_tbl ON language\_id = id;

Notice that instead of separating the two tables using a comma (thereby making it a list), we use the JOIN keyword. The columns that participate in the join condition are preceded by the ON keyword. The WHERE clause can then be used after the join condition specification (ON clause) to specify any further conditions if needed.

The kind of joins where all rows that don't match the join condition exactly are eliminated are called *inner joins*. Thus we can optionally use the full keyword INNER JOIN in our queries without affecting the resultset ([Listing 11-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=534647669&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT author, language FROM authors\_tbl INNER JOIN newlang\_tbl ON language\_id = id;

Resolving Ambiguity in Join Columns

In our example the join condition fields had distinct names – *id* and *language\_id*. But what if in our languages table we kept the key field's name as *language\_id?* This would create an ambiguity in the join condition, which would become the confusing language\_id = language\_id. To resolve this, we need to qualify the column by prepending it by the table name it belongs to and a . (period) ([Listing 11-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=641178214&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE languages\_tbl (language\_id INTEGER, language VARCHAR(20));

INSERT INTO languages\_tbl VALUES (4, 'Tcl');

SELECT author, language FROM authors\_tbl JOIN languages\_tbl ON language\_id = language\_id;

=> ERROR: column reference "language\_id" is ambiguous

SELECT author, language FROM authors\_tbl JOIN languages\_tbl ON authors\_tbl.language\_id = languages\_tbl.language\_id;

| author | language |
| --- | --- |
| Ousterhout | Tcl |

Another way to solve such ambiguity is to qualify the columns using *table aliases*. The concept is to give a short name to a table and then use this to qualify the columns instead of a long, unwieldy table name ([Listing 11-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=641178214&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT author, language FROM authors\_tbl a JOIN newlang\_tbl l ON a.language\_id = l.id;

Here the authors table is given the alias a and the languages table is given the alias l. It is generally considered a good practice to qualify column names of a join condition regardless of whether there is a name ambiguity or not.

Outer Joins

Since we encountered *inner joins* in [Listing 11-3](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=335#335), it gave us a clue to the existence of *outer joins*. In this kind of join, the resultset consists of rows that match the join condition and the rows that don't match the condition from one of the tables. If the rows from the first table that don't match the condition are desired in the resultset, we use a *left outer join*. Otherwise when rows from the second table are required, we use a *right outer join*.

This sounds a bit confusing at first, so let's clarify the concept using an example. Let's add a single row to the *newlang\_tbl* about the Lisp programming language, but we will not make any entry into the authors table for this ([Listing 11-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=311290054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO newlang\_tbl VALUES (6, 'Lisp', 1958, 'ANSI');

If we ran an inner join query on the two tables like [Listing 11-1](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=330#330), we would get a similar output as the query gave that time around. This new row we added would not feature in the resultset. But let's try a left outer join where we explicitly want this new row to be in the results despite not having an entry in the authors table. Our first table must then be the *newlang\_tbl* and the query would be as below ([Listing 11-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=311290054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, author FROM newlang\_tbl n LEFT OUTER JOIN authors\_tbl a ON n.id = a.language\_id;

| language | author |
| --- | --- |
| Prolog | Colmerauer |
| Perl | Wall |
| APL | Ousterhout |
| Tcl | Iverson |
| BASIC | Kemeny |
| BASIC | Kurtz |
| Lisp |  |

Aha, success! The LEFT OUTER JOIN allowed us to sneak the Lisp row into the resultset with a null *author* value. Looking at the query listing, if you immediately think that just by switching the order of the joined tables, we can convert this into a RIGHT OUTER JOIN, then you are absolutely right ([Listing 11-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=311290054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, author FROM authors\_tbl a RIGHT OUTER JOIN newlang\_tbl n ON n.id = a.language\_id;

The output of this query would be exactly the same as in [Listing 11-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=311290054&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#). Notice that our join condition specified in the ON clause did not need any order change. Since all right outer joins can be written as left outer joins (and vice versa), it is rare to find many real-world usages of right outer joins. SQLite goes as far as not support right outer joins at all, which is just as well.

Cross Joins

You might think what would happen if we left out the join condition from our query. Well what happens in the background of running a join query is that first all possible combinations of rows are made from the tables participating in the join. Then the rows that satisfy the join condition are chosen for the output (or further processing). If we leave out the join condition, we get as the output all possible combinations of records ([Listing 11-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=287569109&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This is called a *Cross Join* or *Cartesian Product* of the tables usually denoted by the sign X.

SELECT author, language FROM authors\_tbl, newlang\_tbl;

| author | language |
| --- | --- |
| Colmerauer | Prolog |
| Colmerauer | Perl |
| Colmerauer | APL |
| Colmerauer | Tcl |
| Colmerauer | BASIC |
| Colmerauer | Lisp |
| Wall | Prolog |
| Wall | Perl |
| Wall | APL |
| Wall | Tcl |
| Wall | BASIC |
| Wall | Lisp |
| Ousterhout | Prolog |
| … | … |

The output of the query is truncated here, but when you run it on your computer you should get 36 rows in the result containing each *author* and *language* combination. Another way to rewrite this query is to actually use the JOIN keyword with a preceding argument CROSS as shown below ([Listing 11-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=287569109&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT author, language FROM authors\_tbl CROSS JOIN newlang\_tbl;

Notice the lack of the ON clause, which means no join condition.

What if we were selecting more than one column from the *newlang\_tbl*, say both *language* and *year?* Would the number of combinations increase dramatically from our cross join above? Turns out that no, the number of records in the resultset would be exactly the same as before ([Listing 11-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=287569109&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). A cartesian product is the combination of records from the tables participating in the joins, not within the unit of record for a single table.

SELECT author, language, year FROM authors\_tbl CROSS JOIN newlang\_tbl;

| author | language | year |
| --- | --- | --- |
| Colmerauer | Prolog | 1972 |
| Colmerauer | Perl | 1987 |
| Colmerauer | APL | 1964 |
| … | … | … |

A cross join is not something you would come across often. It is of some utility when either of the tables is small, that is, consisting of a few rows, and you need a combination of all the values of it joined with a bigger table. However I'd advise against running cross joins on actual production database servers unless you really understand why you need them in the scenario.

Self Joins

Sometimes a table within its own columns has meaningful data but one (or more) of its fields refer to another field in the same table. For example, if we have a table in which we capture programming languages that influenced other programming languages and denote the influence relationship by the language id, to show the resolved output we would have to join the table with itself. This is also called a *self join*. Consider the table created below and pay close attention to the data being inserted ([Listing 11-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=730022324&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE inflang\_tbl (id INTEGER PRIMARY KEY,

language VARCHAR(20) NOT NULL,

influenced\_by INTEGER);

INSERT INTO inflang\_tbl (id, language) VALUES (1, 'Fortran');

INSERT INTO inflang\_tbl (id, language, influenced\_by) VALUES (2, 'Pascal', 3);

INSERT INTO inflang\_tbl (id, language, influenced\_by) VALUES (3, 'Algol', 1);

| id | language | influenced\_by |
| --- | --- | --- |
| 1 | Fortran |  |
| 2 | Pascal | 3 |
| 3 | Algol | 1 |

Our goal is to now write a self join query to display which language influenced which one, that is, resolve the *influenced\_by* column ([Listing 11-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=730022324&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT l1.id, l1.language, l2.language AS influenced

FROM inflang\_tbl l1, inflang\_tbl l2

WHERE l1.id = l2.influenced\_by;

| id | language | influenced |
| --- | --- | --- |
| 3 | Algol | Pascal |
| 1 | Fortran | Algol |

Notice the use of table aliases to qualify the join condition columns as separate and the use of the AS keyword that renames the column in the output.

What if we wanted to use the alternative SQL-92 JOIN syntax for our self join? Well as it turns out, there is no special self join keyword or clause because it is not needed. To the SQL query interpreter, you have created an inner join on two tables who just happen to have exactly similar contents. So we can rewrite the [Listing 11-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=730022324&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) query using our familiar JOIN keyword as below ([Listing 11-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=730022324&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT l1.id, l1.language, l2.language AS influenced

FROM inflang\_tbl l1 JOIN inflang\_tbl l2

ON l1.id = l2.influenced\_by;

Non-Equi Joins

The joins we have seen till now have largely dealt with equality in their join condition. While this is the most common way of joining tables together, we are by no means restricted to use only equality. Let's put another join condition between the *newlang\_tbl* and *authors\_tbl* between the *id* and *author\_id* this time ([Listing 11-15](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=902683287&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT id, author\_id, author, language

FROM authors\_tbl, newlang\_tbl WHERE id < author\_id;

| id | author\_id | author | language |
| --- | --- | --- | --- |
| 1 | 2 | Wall | Prolog |
| 1 | 3 | Ousterhout | Prolog |
| 2 | 3 | Ousterhout | Perl |
| 1 | 4 | Iverson | Prolog |
| 2 | 4 | Iverson | Perl |
| 3 | 4 | Iverson | APL |
| 1 | 5 | Kemeny | Prolog |
| 2 | 5 | Kemeny | Perl |
| 3 | 5 | Kemeny | APL |
| 4 | 5 | Kemeny | Tcl |
| 1 | 6 | Kurtz | Prolog |
| 2 | 6 | Kurtz | Perl |
| 3 | 6 | Kurtz | APL |
| 4 | 6 | Kurtz | Tcl |
| 5 | 6 | Kurtz | BASIC |

While not the most logical of results, it does however satisfy our non-equality join condition that in each row the *id* field is lesser than the corresponding *author\_id* field value. We can also freely mix this result with an equality condition using the familiar AND operator within the same query ([Listing 11-16](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=902683287&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT id, author\_id, author, language

FROM authors\_tbl, newlang\_tbl

WHERE id < author\_id AND id = language\_id;

| id | author\_id | author | language |
| --- | --- | --- | --- |
| 3 | 4 | Iverson | APL |
| 5 | 6 | Kurtz | BASIC |

The result now consists of records where the author of a language has their *author\_id* value greater than their created languages' *id*.

### Chapter 12: Subqueries

#### Overview

A *subquery*, simply put, is a query written as a part of a bigger statement. Think of it as a SELECT statement inside another one. The result of the inner SELECT can then be used in the outer query. Let us take a simple example to illustrate this.

Consider the same source tables as the ones in the joins chapter – *authors\_tbl* and *newlang\_tbl*. We will try to write a query (and a subquery) to display the author of a particular language ([Listing 12-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=606908842&resume=yes&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT author FROM authors\_tbl WHERE language\_id IN

(SELECT id FROM newlang\_tbl WHERE language = "Tcl")

| author |
| --- |
| Ousterhout |

The subquery SELECT id FROM newlang\_tbl WHERE language='Tcl' picks the correct language id from the *newlang\_tbl* and passes it on to the outer query on the authors table. This frees us from the responsibility of joining the two tables using the language *id* field.

We can visualize the intermediate step where the subquery has already resolved to a value. The query would now look something like SELECT author FROM authors\_tbl WHERE language\_id IN (4).

Which approach to take in certain situations – a join, a subquery, or a combination of both – is mostly a matter of personal preference. Other times, one approach will be clearly the superior choice. Remember that all joins can be rewritten as subqueries, but the reverse is not true in all cases.

Types of Subqueries

We can broadly classify subqueries into three categories.

1. **Scalar subqueries:** a subquery that returns only a single column of a single row as its output. The example in the previous section, where the subquery returns the id for *Tcl*, is a scalar subquery.
2. **Row subqueries:** a subquery that returns a single row but more than one column. These are the least important type of subqueries since most database management systems do not support it, including SQLite.
3. **Table subqueries:** a table subquery can return more than a single row and many columns per row. In essence, it can return a table itself to take part in your outer query.

To illustrate the usage of table subqueries, let us take an example where we wish to display all the programming language writers who created a language after 1980 ([Listing 12-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=110768086&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT author, language FROM authors\_tbl a,

(SELECT id, language FROM newlang\_tbl WHERE year > 1980) n

WHERE a.language\_id = n.id;

| author | language |
| --- | --- |
| Wall | perl |
| Ousterhout | tcl |

Carefully study the FROM clause of the query above. Our table subquery is placed within it, and it returns a set of languages that were created after 1980. The result consists of two rows and two columns, one of which, that is, *language* is picked up to be displayed in the final output.

Existence Tests in Subqueries

The keyword EXISTS tests the presence of any number of rows returned from a subquery. We usually don't care about the columns being returned by the mere existence of rows satisfying a specific criterion. Let's try to use EXISTS test to display languages who have an author entry in the *authors\_ tbl* ([Listing 12-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=242019001&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM newlang\_tbl

WHERE EXISTS

(SELECT \* FROM authors\_tbl WHERE newlang\_tbl.id = language\_id)

| language | year |
| --- | --- |
| Prolog | 1972 |
| Perl | 1987 |
| APL | 1964 |
| Tcl | 1988 |
| BASIC | 1964 |

Notice the subquery WHERE clause in this case. It is effectively referencing the outer table field using newlang\_tbl.id. For whichever languages this existence test will be satisfied, the outer query will add to the resultset.

We can add the option NOT to the existence test to find the complement of the result ([Listing 12-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=242019001&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language, year FROM newlang\_tbl

WHERE NOT EXISTS

(SELECT \* FROM authors\_tbl WHERE newlang\_tbl.id = language\_id)

| language | year |
| --- | --- |
| Lisp | 1958 |

Recall that we had never put the corresponding entry in the authors table for Lisp in the last chapter.

So who created Lisp anyway?

Lisp is the second oldest programming language whose major dialect is still in active use. John McCarthy created Lisp in 1958 as a part of his research, and other people chipped in to help implement it on the computers of that era.

McCarthy by all accounts was a genius-level intellect, widely admired by his peers. He was one of the pioneers of the field of Artificial intelligence and even coined the term. With the creation of Lisp, he advanced the field of programming language design by leaps and bounds. Over the past three decades, features from Lisp are slowly trickling into mainstream programming languages. Many renowned technologists still marvel at the design of the decades-old Lisp dialects – Common Lisp and Scheme.

Though McCarthy died in 2011 at the age of 84, his legacy and work lives on.

Using Subqueries in INSERT Statements

We can even use subqueries inside other SQL statement like INSERT. Let us try to add a new language and a new author in our tables and ease our task of remembering *id* numbers by just a bit by using subqueries ([Listing 12-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=295406751&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO newlang\_tbl (id, language, year, standard) VALUES

(7, 'Pascal', 1970, 'ISO');

The updated content of our programming languages table now looks as shown below ([Table 12-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=295406751&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

| id | language | year | standard |
| --- | --- | --- | --- |
| 1 | Prolog | 1972 | ISO |
| 2 | Perl | 1987 |  |
| 3 | APL | 1964 | ANSI |
| 4 | Tcl | 1988 |  |
| 5 | BASIC | 1964 | ANSI |
| 6 | Lisp | 1958 | ANSI |
| 7 | Pascal | 1970 | ISO |

While inserting a new entry into the *authors\_tbl*, we can either remember that we used the *language\_id* as 7 for Pascal or use a subquery. Let us see an example of the latter approach ([Listing 12-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=295406751&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). After all, the title of the chapter gave away our approach!

INSERT INTO authors\_tbl (author\_id, author, language\_id) VALUES

(7, 'Wirth', (SELECT id FROM newlang\_tbl WHERE language="Pascal"))

We believe that this should put the correct language id for Mr. Wirth since he created Pascal. Let us verify this belief by looking at the contents of the table.

| author\_id | author | language\_id |
| --- | --- | --- |
| 1 | Colmerauer | 1 |
| 2 | Wall | 2 |
| 3 | Ousterhout | 4 |
| 4 | Iverson | 3 |
| 5 | Kemeny | 5 |
| 6 | Kurtz | 5 |
| 7 | Wirth | 7 |

You can even use subqueries to control your UPDATE and DELETE statements. The logic remains much the same as with using subqueries in SELECT and INSERT.

Using ANY and ALL

The ANY operator used with the arithmetic comparison operators can be used to check a column value in comparison to a similar value(s) generated in the subquery. For example, if we wanted to display all the languages but exclude the oldest one from the result, we could combine > and ANY to achieve this ([Listing 12-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=752375157&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language FROM newlang\_tbl WHERE year > ANY (SELECT year FROM newlang\_tbl)

| language |
| --- |
| Prolog |
| Perl |
| APL |
| Tcl |
| BASIC |
| Pascal |

Only Lisp does not have a creation year that is not greater than *any* of the list of values returned by the subquery. Obviously, this is because the smallest value returned is the creation year of Lisp itself, and thus it does not feature in the final result.

Now what would happen if we reversed our comparison operator to < ANY? The result would include all languages whose *year* value is less than any one of the creation years returned by the subquery ([Listing 12-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=752375157&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language FROM newlang\_tbl WHERE year < ANY (SELECT year FROM newlang\_tbl)

| language |
| --- |
| Prolog |
| Perl |
| APL |
| BASIC |
| Lisp |
| Pascal |

We notice that Lisp has snuck into the resultset but Tcl is notably absent. This is because the *year* of Tcl, that is, 1988 is not less than any of the values returned by the subquery. Equal to? Sure, but not distinctly less than.

The other comparison conjunction we can use with ANY is =, but that is rarely seen because it is equivalent to using IN (), which is much more intuitive.

SQLite does not support ANY or ALL operators

If you tried running the above examples in SQLite, you would get an error message as below.

Error: near "SELECT": syntax error

SQLite currently does not support these keywords, but we can still achieve the same results using what we have to work with. Let's attempt to rewrite [Listing 12-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=752375157&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) displaying all languages but the oldest one.

SELECT language FROM newlang\_tbl WHERE year <> (SELECT MIN(year) FROM newlang\_tbl)

The above query computes our desired resultset just fine and is pretty readable. Some, including yours truly, actually prefer it to the ANY syntax. if you are wondering about <>, it means not equal to.

The ALL operator works similarly, but the value in the WHERE clause must hold true for *all* of the values returned from the subquery ([Listing 12-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=752375157&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). One scenario where ALL gets usage is to find data related to extreme values like minima and maxima. You are of course free to choose the built-in functions MAX and MIN for the purpose too.

SELECT language FROM newlang\_tbl WHERE year <= ALL (SELECT year FROM newlang\_tbl)

| language |
| --- |
| Lisp |

Only Lisp being the oldest language in our table would satisfy the ALL criteria of having a *year* value less than or equal to all the values from the subquery. Similarly, we can use ALL to find the latest language too ([Listing 12-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=752375157&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT language FROM newlang\_tbl WHERE year >= ALL (SELECT year FROM newlang\_tbl)

| language |
| --- |
| Tcl |

### Chapter 13: Working in Sets

#### Overview

Set theory is a branch of discrete mathematics that deals with a collection of objects. There is a lot of conceptual overlap between set theory and relational database concepts. It is no wonder that the output of a query is frequently called a result**set**.

Primitive set theoretic operations like *union, intersection*, and *difference* are increasingly supported in various implementations. We will now explore the theory behind these operations and how to use them in SQL.

Union

The *union* is an operation that combines elements of two sets. Let's say we have the following two sets consisting of a bunch of numbers ([Listing 13-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=757033688&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

set1 = { 1, 3, 5 }

set2 = { 1, 2, 3 }

The resulting union set will be a set consisting of all of these elements repeated exactly once, that is, no duplicates are allowed ([Listing 13-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=757033688&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Note that the order of a set is unimportant. Think of it as a bag of elements rather than an ordered collection.

set1 UNION set2 = { 1, 3, 5, 2 }

Let's now look at how to use simulate the union operation in SQL. Consider our programming languages table and its data below as we last left it in [Chapter 10](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=285#285) ([Table 13-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=757033688&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 3 | APL | Iverson | 1964 | ANSI |
| 4 | JOVIAL | Schwartz | 1959 | US-DOD |
| 5 | APT | Ross | 1959 | ISO |
| 6 | PL/I | IBM | 1964 | ECMA |
| 7 | Tcl | Ousterhout | 1988 |  |
| 8 | Fortran | Backus | 1957 | ANSI |

If we wanted to get the list of creation years of languages standardized by either ANSI or ISO, we could use a UNION keyword to achieve this ([Listing 13-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=757033688&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT year FROM proglang\_tbl WHERE standard = 'ANSI'

UNION

SELECT year FROM proglang\_tbl WHERE standard = 'ISO'

| year |
| --- |
| 1959 |
| 1957 |
| 1964 |
| 1972 |

Since we had four entries in our table with a *standard* value as ANSI or ISO, we got our expected four rows in the resultset. Note that there were no duplicate entries to be processed. But what if there were duplicate entries to process with the UNION operation ([Listing 13-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=757033688&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#))?

SELECT standard FROM proglang\_tbl WHERE language = 'Fortran'

UNION

SELECT standard FROM proglang\_tbl WHERE language = 'APL'

| standard |
| --- |
| ANSI |

Both the languages we specified in our WHERE clause were standardized by ANSI. The UNION operation, just like in discrete maths, removed the duplicated value and gave out a single row as the result.

There is another related SQL operation UNION ALL that will simulate the act of combination but will not eliminate duplicates ([Listing 13-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=757033688&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). The advantage you get by using this is performance improvement since the SQL engine does not have to bother with checking for duplicates. If you have constructed your participating queries in such a way that there are no repeated values, using a UNION ALL would improve your query processing time.

SELECT standard FROM proglang\_tbl WHERE language = 'Fortran'

UNION ALL

SELECT standard FROM proglang\_tbl WHERE language = 'APL'

| standard |
| --- |
| ANSI |
| ANSI |

Intersection

The *intersection* operation outputs only the common elements in the input sets. If we apply an intersection to the two sets in the previous section, we get a resulting set of two elements ([Listing 13-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=382239918&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

set1 INTERSECTION set2 = { 1, 3 }

As with union, each common value is displayed only once. Duplicates are removed from the final result set.

Translating this to SQL is pretty simple; instead of using UNION we use the keyword INTERSECT to get common elements ([Listing 13-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=382239918&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT standard FROM proglang\_tbl WHERE year = 1964

INTERSECT

SELECT standard FROM proglang\_tbl WHERE year = 1957

| standard |
| --- |
| ANSI |

Something to keep in mind here is that the INTERSECT operator would find the exact common values between the two queries that precede and succeed it. That means the entire records of the result and not just common values from a part of it. While in the previous example, our result set had only one column to be given back – *standard*, let's see what happens when we add another column to the result list ([Listing 13-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=382239918&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT year, standard FROM proglang\_tbl WHERE year = 1964

INTERSECT

SELECT year, standard FROM proglang\_tbl WHERE year = 1957

=> (0 rows)

The output is no rows at all. The first query would select records for PL/I and APL while the second for Fortran. But all these languages have a different combined value of *(year, standard)*, giving us a net zero result.

Note that while the ANSI SQL standard does provision for an INTERSECT ALL operator, I'm yet to come across a database management system that implements it. PostgreSQL happily ignores that you wrote the ALL clause and simply gives back an INTERSECT result.

Difference

The *difference* operation between sets, written as set1 – set2 is a list of all elements in *set1* that do not occur in *set2* ([Listing 13-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=172170627&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). If an element is only in *set2*, it will not be captured by the plain difference operation.

set1 DIFFERENCE set2 = { 5 }

set2 DIFFERENCE set1 = { 2 }

Let's try and write a SQL statement to emulate this logic with our familiar IN and NOT IN operators. But first let's insert a row into our table so that we can see the difference operation in action ([Listing 13-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=172170627&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tbl (id, language, author, year, standard) VALUES

(9, 'RPG', 'IBM', 1964, 'ISO')

Suppose we wish to list out the years of creation of languages that were standardized by ISO but not the ANSI ([Listing 13-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=172170627&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). From our source table, we find that three languages were standardized by ISO with years 1972, 1959, and 1964. But since in 1964, APL was created, which was eventually standardized by ANSI, we should ideally be left with the answer 1972 and 1959.

SELECT year FROM proglang\_tbl WHERE standard IN ('ISO')

AND standard NOT IN ('ANSI')

| year |
| --- |
| 1972 |
| 1959 |
| 1964 |

Whoa, what sorcery is this!?! We thought 1964 would be ineligible because of ANSI standardization. But clearly this is not the case. What has happened actually is that first there was a scan of ISO rows – giving us three values. Then ANSI rows were discounted but not necessarily from the first result but the table as a whole. So while the APL 1964 was left off, the freshly inserted RPG 1964 still remained, effectively making our second condition worthless. The correct way to achieve this is using the set difference operator EXCEPT as below ([Listing 13-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=172170627&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT year FROM proglang\_tbl WHERE standard IN ('ISO')

EXCEPT

SELECT year FROM proglang\_tbl WHERE standard IN ('ANSI')

| year |
| --- |
| 1972 |
| 1959 |

Voila, this seems to yield the correct answer! If you happen to be using an Oracle system, replace EXCEPT with MINUS to achieve the exact same result.

When we write more than a single SELECT as a part of a single query and join them using a set theoretic operator, such statements are called *compound queries*. Do note that many database management systems restrict the use of compound queries as subqueries. Sybase Adaptive Server Enterprise is one such popular DBMS that doesn't allow you to write a UNION inside a subquery.

-----------------------------------------------------------------------------------------

## Domain

### Pre-Assessment

Q1: Inner Join (ID 27852) = V

Which statement best defines an inner join?

A join in which records will only display if the value of the join field is in both tables

A join in which records will be displayed from one table specified in the join field

A join in which records are filtered from one table and displayed on the second table

A join in which all records are displayed from one table, and only the value of the join field is displayed from the second table

**A join in which records will only display if the value of the join field is in both tables**

Q2: Full Outer Join (ID 27853) = V

When would one normally implement a full outer join? Choose two answers.

Only matching records from both tables in a join need to be visible

After combining two data sets into one

Mismatched records from both tables in a join need to be visible

Before combining two data sets into one

**Mismatched records from both tables in a join need to be visible**

**After combining two data sets into one**

Q3: Self Join (ID 27854) = V

Which statement best defines a self join?

A join between two rows in the same table

A join between two tables

A join between two columns in the same table

A join between two databases

**A join between two columns in the same table**

Q4: Table Alias (ID 27855) = V

Once a table alias is established, it has to be used throughout the \_\_\_.

Row

Column

Query

Database

**Query**

Q5: Sorting Clause (ID 27856) = V

What clause is used to sort information in a query?

SORT BY

ORDER BY

SORT

ORDER FROM

**ORDER BY**

Q6: WHERE Clause (ID 27857) = V

The WHERE clause creates \_\_\_ on the data being queried.

Tags

Notes

Filters

Lists

**Filters**

Q7: Percent Symbol (ID 27858) = V

The percent symbol (%) is a \_\_\_ that allows any number of characters.

Wildcard

Clause

Statement

Table

**Wildcard**

Q8: Join Alternative (ID 27859) = V

Rather than using a join, one may use a \_\_\_ to display fields from a corresponding table.

Substitute query

SELECT statement

Query

Subquery

**Subquery**

Q9: Records Before a Date (ID 27860) = V

Which comparison operator should one use to return records from before a given date?

>

<

>=

<=

**<**

Q10: WHERE and GROUP BY Clauses (ID 27861) = V

Where should a WHERE clause be used within the GROUP BY clause?

Before the GROUP BY clause

After the GROUP BY clause

With the GROUP BY clause

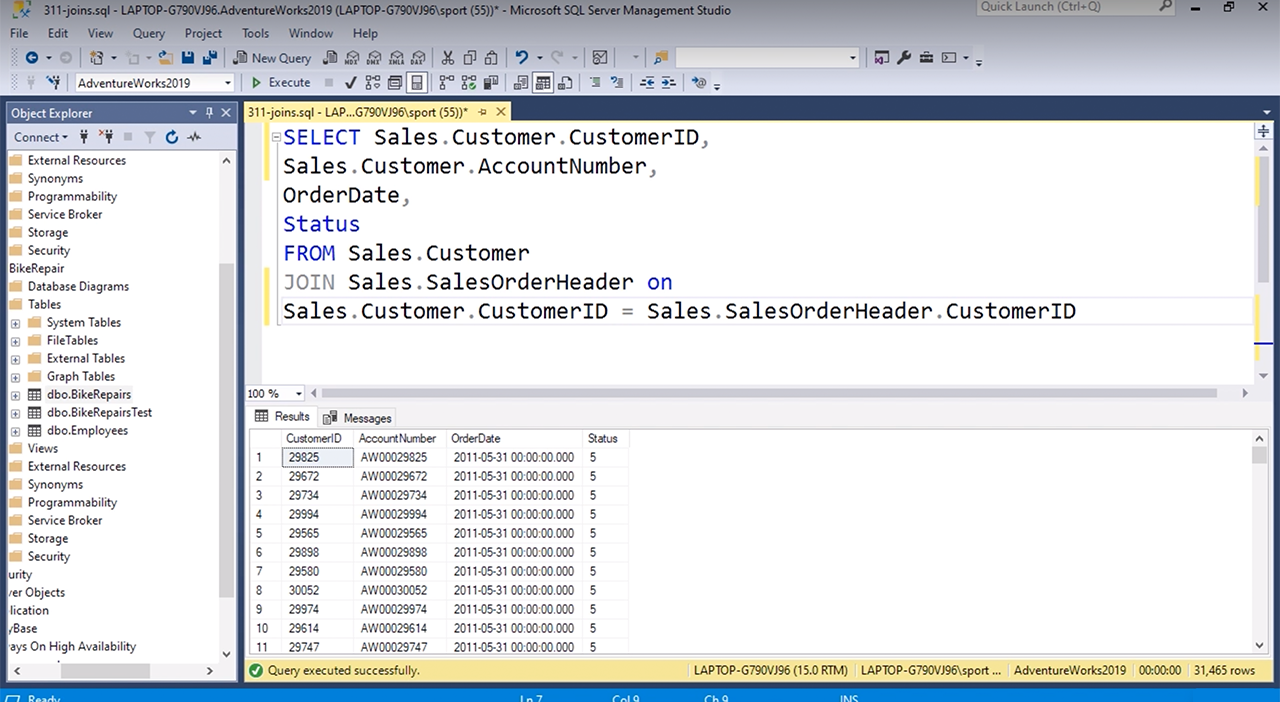
The WHERE clause is not used with the GROUP BY clause

**Before the GROUP BY clause**

### Post-Assessment

Q1: Joins (ID 27862) = V

Refer to the image. What type of join is shown?



Inner Join

Left Outer Join

Right Outer Join

Full Outer Join

**Inner Join**

Q2: Joins (ID 27863) = V

A user wants to see all records from the Sales.Customer table and only records with an entry in the CustomerID column from the Sales.SalesOrderHeader table. The Sales.Customer table is listed first in the SQL Query Window. What type of join should the user implement?

Inner join

Left outer join

Right outer join

Full outer join

**Left Outer Join**

Q3: Cartesian Products (ID 27864) = V

Which statement best defines a cartesian product, also known as a cross join?

No records from either table in a join are related to one another

Specified records in one table are related to specified records in another

Specified records in one table are related to every record in another

Every record in one table is related to every record in another

**Every record in one table is related to every record in another**

Q4: Self Joins (ID 27865) = V

Refer to the image. What will happen when this self join is executed?



The result will display the employees with a manager but omit the employee without a manager

The result will display all employees

The result will display the employee without a manager but omit the employees with a manager

An error will occur

**The result will display the employees with a manager but omit the employee without a manager**

Q5: UNION (ID 27866) = V

What keyword should one use between two SELECT statements to combine two datasets into one?

UNION

MATCH

MERGE

INTERSECT

**UNION**

Q6: UNION (ID 27867) = V

What keyword should one use between two SELECT statements to show records that are alike in two tables?

UNION

MATCH

MERGE

INTERSECT

**INTERSECT**

Q7: DISTINCT (ID 27868) = V

What purpose does a DISTINCT keyword after a SELECT statement serve?

It distinguishes between two different types of tables

It prevents duplicate tables from showing in a result

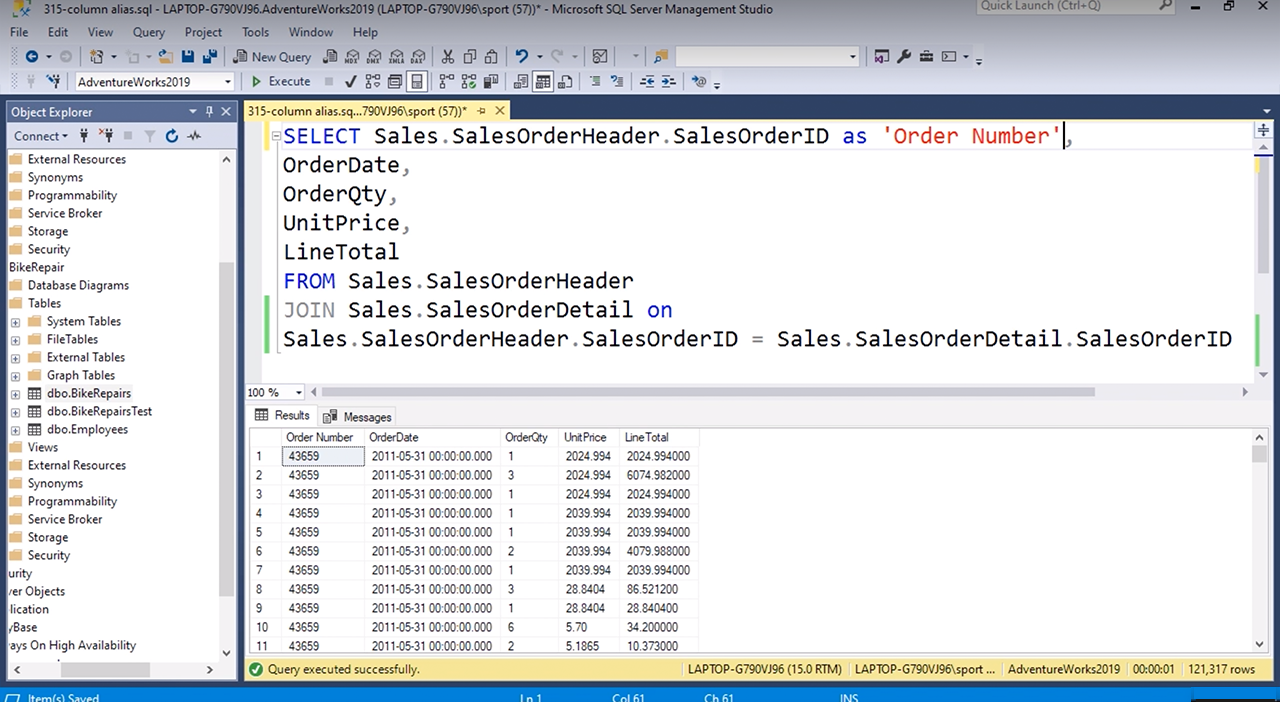
It prevents duplicate rows from showing in a result

It prevents duplicate columns from showing in a result

**It prevents duplicate rows from showing in a result**

Q8: Column Alias (ID 27869) = V

Refer to the image. What is the column alias?

****

Sales.SalesOrderHeader

Order Number

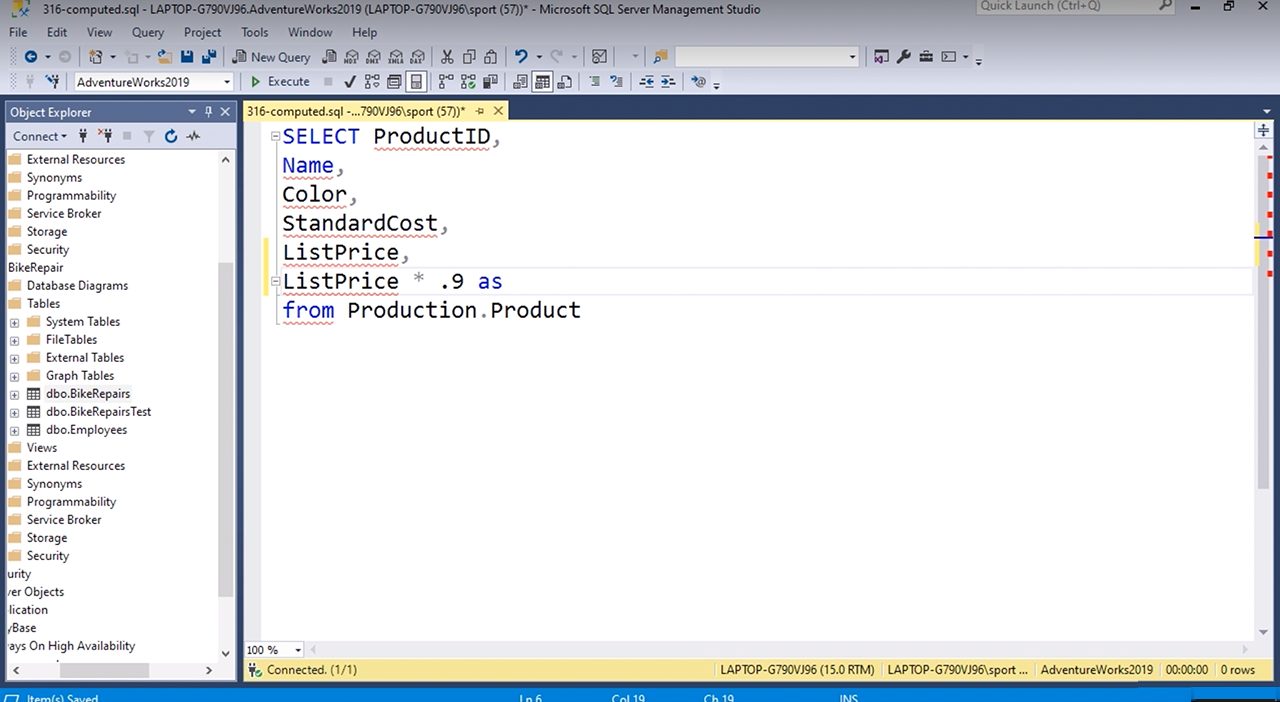
LineTotal

Sales.SalesOrderDetail

**Order Number**

Q9: Adding New Column (ID 27870) = V

Refer to the image. A new column is being added to the table. What information will the new column show?



The list price

The list price with 10% taken off

The list price with 90% taken off

ProductID

**The list price with 10% taken off**

Q10: Computed Column (ID 27871) = V

What function can be added to a computed column to show the results as dollar amounts?

Currency function (cur)

Decimal function (dec)

Dollar function (dol)

String function (str)

**Dollar function (dol)**

**Currency function (cur)**

Q11: Reverse Order (ID 27872) = V

What letters can be added to the end of a column to show the column in reverse order?

DESC

REV

BACK

DOWN

**DESC**

Q12: WHERE Clause (ID 27873) = V

In a query involving a table, where does the WHERE clause belong?

Before the table

After the table

Within the table

Outside the table

**After the table**

Q13: LIKE Clause (ID 27874) = V

What clause should be used to get an approximate match to a search rather than an exact match?

APPROXIMATE clause

SIMILAR clause

RELEVANT clause

LIKE clause

**LIKE clause**

Q14: WHERE Clause (ID 27875) = V

What keyword can be used with a WHERE clause to filter information within a range?

FROM

MIDDLE

BETWEEN

RANGE

**BETWEEN**

Q15: AND Clause (ID 27876) = V

What keyword can be used between two criteria when using a WHERE clause to narrow results further?

BUT

NARROW

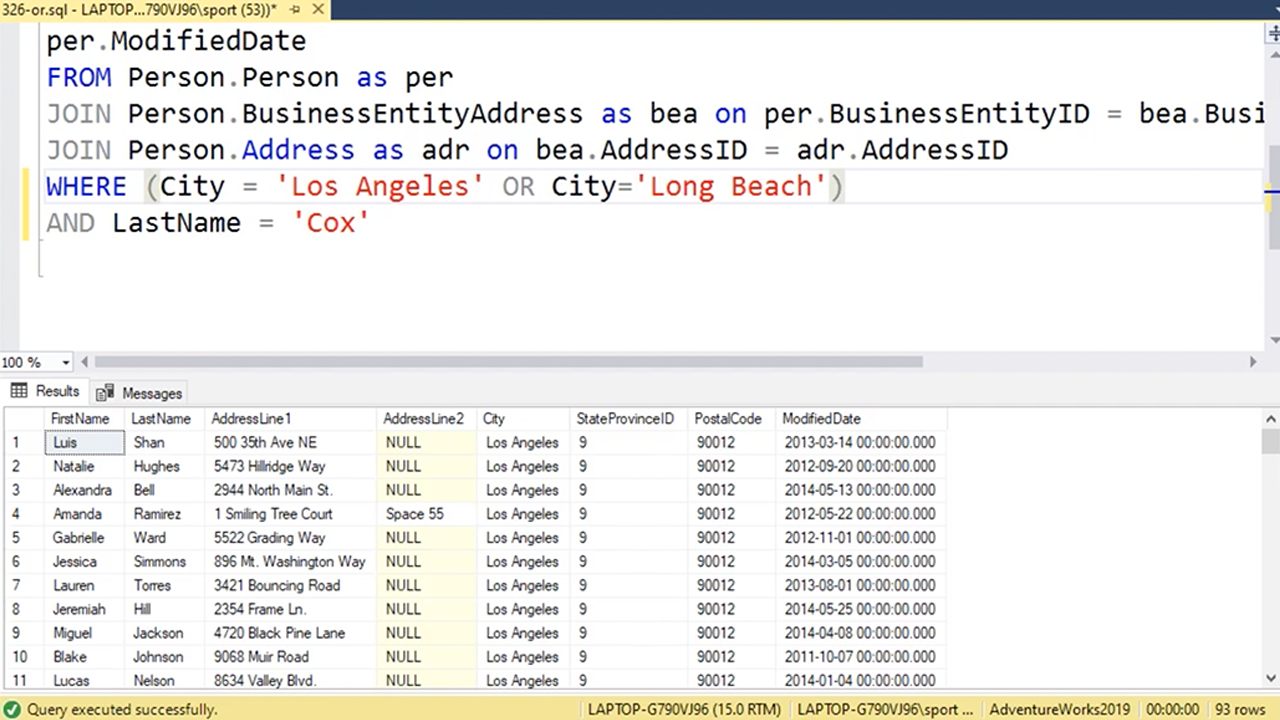
RESTRICT

AND

**AND**

Q16: AND Clause (ID 27877) = V

Refer to the image. What will the results of this query show?



People living in Los Angeles with the last name Cox

People living in Los Angeles with the last name Cox and people living in Long Beach with the last name Cox

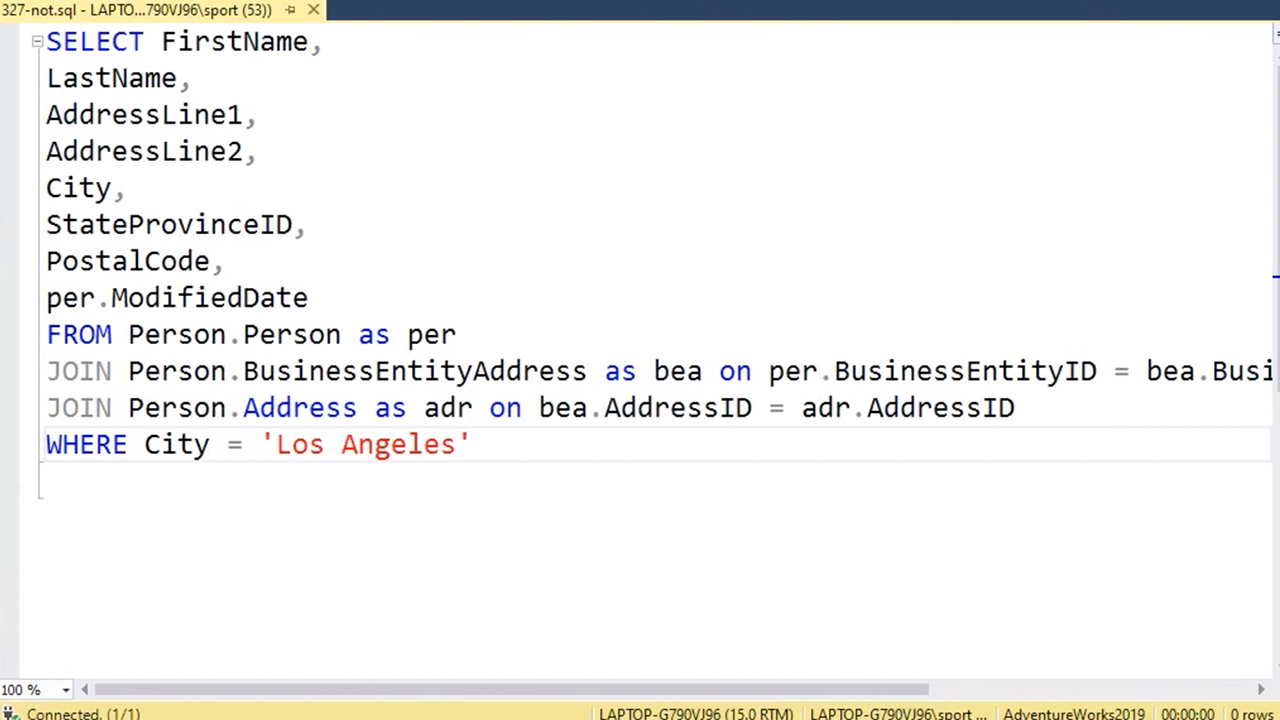
People living in Long Beach with the last name Cox and people living in Los Angeles

People living in Los Angeles with the last name Cox and people living in Long Beach

**People living in Los Angeles with the last name Cox and people living in Long Beach with the last name Cox**

Q17: WHERE & NOT Clauses (ID 27878) = V

Refer to the image. Which WHERE clause will show every person except for those living in Los Angeles in the results?



WHERE NOT City = 'Los Angeles'

NOT WHERE City = 'Los Angeles'

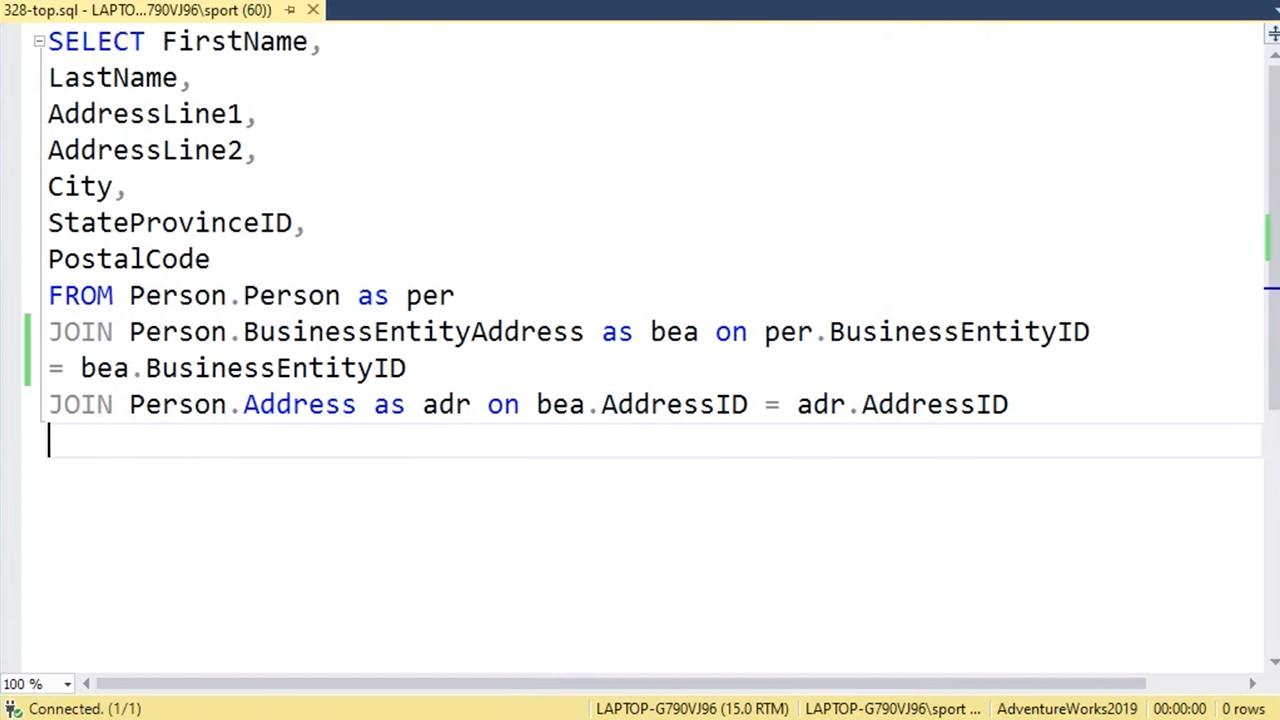
WHERE City = NOT 'Los Angeles'

WHERE City = 'NOT Los Angeles'

**WHERE NOT City = ‘Los Angeles’**

Q18: TOP Clause (ID 27879) = V

Refer to the image. What should the first line of the SELECT statement read to limit results to the first 100 records in the table?



SELECT FirstName, TOP 100

TOP 100 SELECT FirstName,

SELECT TOP 100 FirstName,

SELECT FirstName TOP 100,

**SELECT TOP 100 FirstName,**

Q19: WHERE Clause (ID 27880) = V

What keyword or key phrase can be used with a WHERE clause to return records matching multiple criteria?

INCLUDE

NOT IN

IN

ON

**IN**

Q20: WHERE Clause (ID 27881) = V

What keyword or key phrase can be used to return only records that do not match multiple criteria?

INCLUDE

NOT IN

IN

ON

**NOT IN**

Q21: ANY Clause (ID 27882) = V

What keyword shows all records with at least one match in a subquery?

ANY

ALL

FULL

SUB

**ANY**

Q22: WHERE Clause (ID 27883) = V

What keyword shows every record that matches the same criteria?

ANY

ALL

FULL

SAME

**ALL**

Q23: NULL Clause (ID 27884) = V

What keyword or key phrase shows records with empty fields?

VACANT

EMPTY

NULL

NOT NULL

**NULL**

Q24: NULL Clause (ID 27885) = V

What keyword or key phrase shows records with fields that are not empty?

OCCUPIED

FULL

NULL

NOT NULL

**NOT NULL**

Q25: Comparison Operators (ID 27886) = V

Which comparison operator returns records on or before a given date?

>

<

>=

<=

**<=**

Q26: Comparison Operators (ID 27887) = V

Which comparison operator returns records on or after a given date?

>

<

>=

<=

**>=**

Q27: Comparison Operators (ID 27888) = V

Which comparison operator returns records after a given date?

>

<

>=

<=

**>**

Q28: Aggregate Functions (ID 27889) = V

Which aggregate totals the values of a field?

GROUP BY

ADD

SUM

TOTAL

**SUM**

Q29: Aggregate Functions (ID 27890) = V

What clause is used for columns that are not being used in an aggregate?

GROUP BY

COLLECT BY

SUM

EXCLUDE BY

**GROUP BY**

Q30: Aggregate Functions (ID 27891) = V

What keyword or key phrase is used to filter records from an aggregate?

FILTER

AGG FILTER

INCLUDING

HAVING

**HAVING**

Q31: Aggregate Functions (ID 27892) = V

Which aggregate is used to show the lowest value in a numeric field within a group?

LOW

MIN

HIGH

MAX

**MIN**

Q32: Aggregate Functions (ID 27893) = V

Which aggregate is used to show the highest value in a numeric field within a group?

LOW

MIN

HIGH

MAX

**MAX**

Q33: Aggregate Functions (ID 27894) = V

Which aggregate is used to show the number of records for each item in a group?

NUM

COUNT

TOTAL

AVG

**COUNT**

Q34: Aggregate Functions (ID 27895) = V

Which aggregate is used to show the average of a numerical field?

MID

MEAN

AVG

AVE

**AVG**

## Videos

### Inner Joins - Introduction (5m)

Right-click Views, Add View, Choose Two Tables, click OK

Automatic Relationship Between Customers & Orders

CustomerID PK -> CustomerID FK

### Using Cross Joins (7m)

DONE

### Outer Joins (8m)

DONE

### UNION Operator and Combining Results (5m)

TBC

### Filter Results with WHERE Clause (9m)

TBC

### Ordering Results with ORDER BY (6m)

TBC

### AVG and GROUP BY (8m)

TBC

### Using HAVING Clause with Queries (5m)

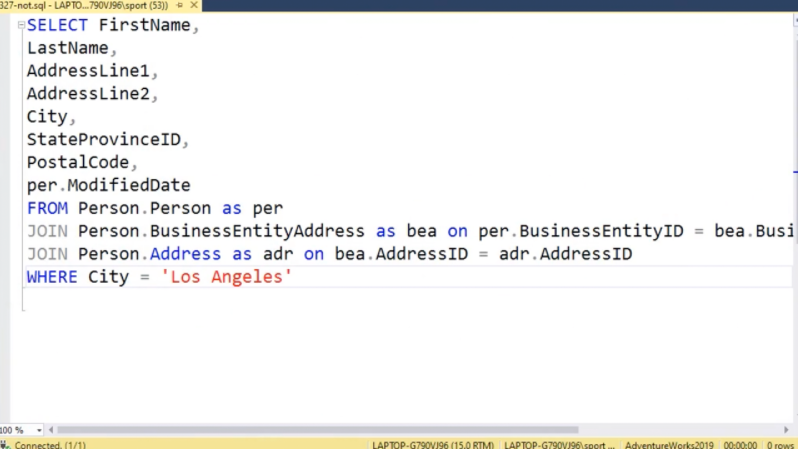
TBC

## Assignment

Please upload screenshots of your solutions for any 2 of 5 Exercises:

## Quiz

Q1: Refer to the image. Which WHERE clause will show every person except for those living in Los Angeles in the results?



NOT WHERE City = ‘Los Angeles’

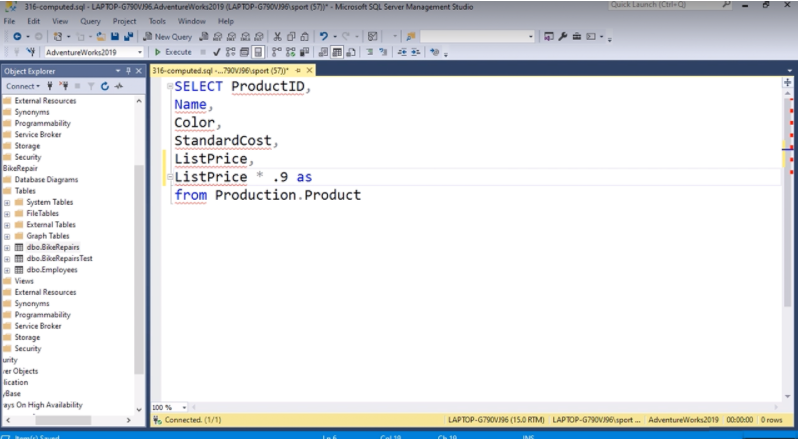
WHERE City = NOT ‘Los Angeles’

WHERE NOT City = ‘Los Angeles’

WHERE City = ‘Not Los Angeles’

**WHERE NOT City = ‘Los Angeles’**

Q2: Refer to the image. A new column is being added to the table. What information will the new column show?



The list price

ProductID

The list price with 10% taken off

The list price with 90% taken off

**The list price with 10% taken off**

Q3: Which statement best defines a cartesian product?

Specified records in one table are related to specified records in another

Specified records in one table are related to every record in another

No records in either table in a join are related to one another

Every record in one table is related to every record in another

**Every record in one table is related to every record in another**

Q4: What keyword can be used with a WHERE clause to filter information within a range?

RANGE

MIDDLE

FROM

BETWEEN

**BETWEEN**

Q5: What keyword should one use between two SELECT statements to combine two datasets into one?

MERGE

MATCH

INTERSECT

UNION

**UNION**

Q6: Which comparison operator returns records on or before a given date?

>=

<

>

<=

**<=**

Q7: Which aggregate is used to show the average of a numerical field?

MID

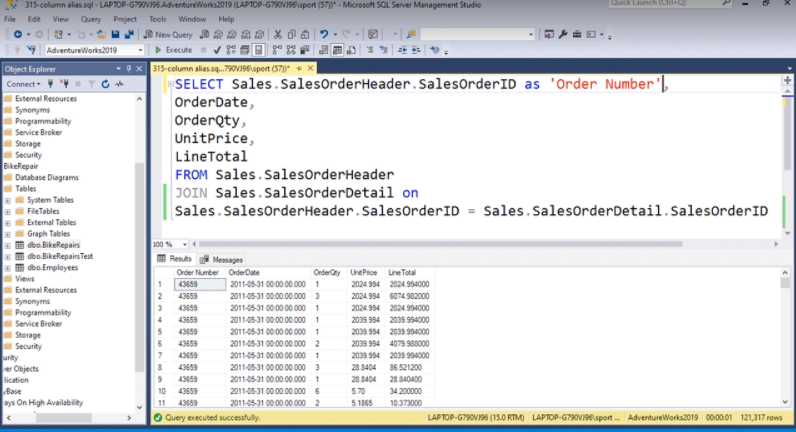
MEAN

AVE

AVG

**AVG**

Q8: Refer to the image. What is the column alias?



Order Number

Sales.SalesOrderDetail

Sales.SalesOrderHeader

LineTotal

**Order Number**

Q9: What keyword should one use between two SELECT statements to show records that are alike in two tables?

UNION

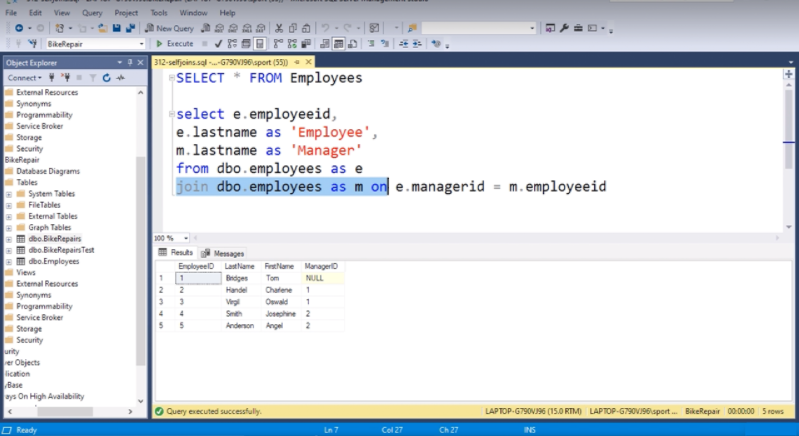
MERGE

INTERSECT

MATCH

**INTERSECT**

Q10: Refer to the image. What will happen when this self join is executed?



An error will occur

The result will display the employee without a manager but omit the employees with a manager

The result will display the employees with a manager but omit the employee without a manager

The result will display all employees

**The result will display the employees with a manager but omit the employee without a manager**

Q11: What keyword or key phrase can be used with a WHERE clause to return records matching multiple criteria?

NOT IN

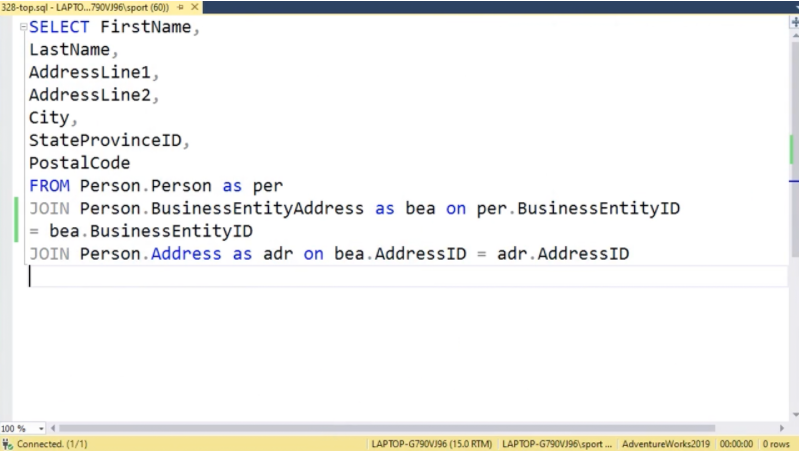
ON

INCLUDE

IN

**IN**

Q12: Refer to the image. What should the first line of the SELECT statement read to limit results to the first 100 records in the table?



SELECT FirstName TOP 100,

SELECT FirstName, TOP 100

SELECT TOP 100 FirstName,

TOP 100 SELECT FirstName,

**SELECT TOP 100 FirstName**

Q13: What keyword can be used between two criteria when using a WHERE clause to narrow results further?

NARROW

AND

RESTRICT

BUT

**AND**

Q14: What keyword shows all records with at least one match in a subquery?

SUB

FULL

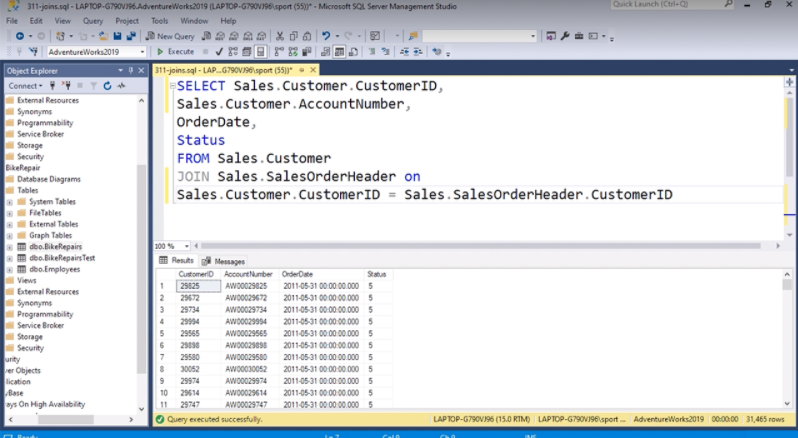
ALL

ANY

**ALL**

**ANY**

Q15: Refer to the image. What type of join is shown?



Right Outer Join

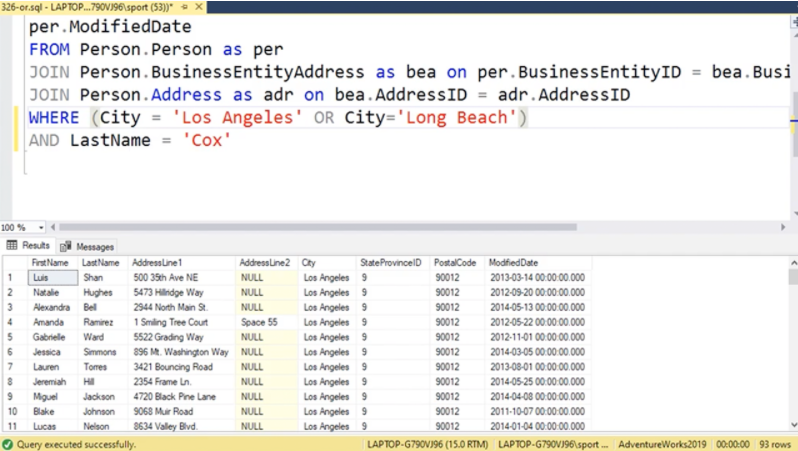
Left Outer Join

Full Outer Join

Inner Join

**Inner Join**

Q16: Refer to the image. What will the results of this query show?



People living in Los Angeles with the last name Cox and people living in Long Beach

People living in Los Angeles with the last name Cox

People living in Long Beach with the last name Cox and people living in Los Angeles

People living in Los Angeles with the last name Cox and People living in Long Beach with the last name Cox

**People living in Los Angeles with the last name Cox and People living in Long Beach with the last name Cox**

Q17: What clause should be used to get an approximate match to a search rather than an exact match?

APPROXIMATE clause

SIMILAR clause

LIKE clause

RELEVANT clause

**LIKE clause**

Q18: What clause is used for columns that are not being used in an aggregate?

EXCLUDE BY

COLLECT BY

GROUP BY

SUM

**GROUP BY**

Q19: What purpose does a DISTINCT keyword after a SELECT statement serve?

It prevents duplicate columns from showing in a result

It prevents duplicate tables from showing in a result

It distinguishes between two different types of tables

It prevents duplicate rows from showing in a result

**It prevents duplicate rows from showing in a result**

Q20: What keyword or key phrase shows records with fields that are not empty?

NULL

OCCUPIED

NOT NULL

FULL

**NOT NULL**

Q21: What keyword or key phrase shows records with empty fields?

NOT NULL

EMPTY

VACANT

NULL

**NULL**

Q22: Which aggregate is used to show the lowest value in a numeric field within a group?

HIGH

MIN

MAX

LOW

**MIN**

Q23: Which aggregate is used to show the number of records for each item in a group?

NUM

COUNT

TOTAL

AVG

**COUNT**

Q24: Which comparison operator returns records after a given date?

>=

<

>

<=

**>**

Q25: Which function can be added to a computed column to show the results as dollar amounts?

Decimal function (dec)

String function (str)

Currency function (cur)

Dollar function (dol)

**Currency function (cur)**

**String function (str)**

Q26: Which comparison operator returns records on or after a given date?

>=

<

>

<=

**>=**

Q27: What keyword shows every record that matches the same criteria?

FULL

ALL

SAME

ANY

FULL

ALL

Q28: What keyword or key phrase is used to filter records from an aggregate?

FILTER

HAVING

AGG FILTER

INCLUDING

**HAVING**

Q29: In a query involving a table, where does the WHERE clause belong?

Outside the table

Within the table

Before the table

After the table

**After the table**

Q30: Which aggregate is used to show the highest value in a numeric field within a group?

LOW

MAX

MIN

HIGH

**MAX**

Q31:What letters can be added to the end of a column to show the column in reverse order?

DOWN

REV

DESC

BACK

**DESC**

Q32: A user wants to see all records from the Sales.Customer table and only records with an entry in the CustomerID column from the Sales.SalesOrderHeader table.

The Sales.Customer table is listed first in the SQL Query Window.

What type of join should the user implement?

Right Outer Join

Full Outer Join

Inner Join

Left Outer Join

**Left Outer Join**

Q33: Which aggregate totals the values of a field?

TOTAL

ADD

GROUP BY

SUM

**SUM**